Improving Postgres' Efficiency using JIT and other techniques

Andres Freund

PostgreSQL Developer & Committer

Email: andres@anarazel.de

Email: andres.freund@enterprisedb.com

Twitter: @AndresFreundTec

anarazel.de/talks/scale16x-2018-03-09/jit-and-other-efficiency.pdf

Motivation

TPC-H Q01

```
SELECT
    l_returnflag,
    l_linestatus,
    sum(l_quantity) AS sum_qty,
    sum(l_extendedprice) AS sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
    avg(l_quantity) AS avg_qty,
    avg(l_extendedprice) AS avg_price,
    avg(l_discount) AS avg_disc,
    count(*) AS count_order
FROM lineitem
WHERE l_shipdate <= date '1998-12-01' - interval '74 days'
GROUP BY l_returnflag, l_linestatus
ORDER BY l_returnflag, l_linestatus;
```

```
Samples: 87K of event 'cycles:ppp', cnt (approx.): 71706618234
  Overhead  Command   Shared Object     Symbol
-   35.96%  postgres  postgres          [.] ExecInterpExpr
      + 72.86% ExecAgg
      - 18.33% tuplehash_insert
           LookupTupleHashEntry
           ExecAgg
           ExecSort
      + 8.81% ExecScan
-   10.79%  postgres  postgres          [.] slot_deform_tuple
        slot_getsomeattrs
      - ExecInterpExpr
         + 77.31% ExecScan
         + 22.69% tuplehash_insert
+    4.96%  postgres  postgres          [.] tuplehash_insert
+    4.53%  postgres  postgres          [.] float8_accum
+    3.21%  postgres  postgres          [.] float8pl
+    2.61%  postgres  postgres          [.] bpchareq
+    2.40%  postgres  postgres          [.] hashbpchar
```

What is “Just In Time” compilation

- Convert forms of “interpreted” code into native code
- Specialize code for specific constant arguments
- Achieve speedups via:
  - reduced total number of instructions
  - reduced number of branches
  - reduced number of indirect jumps / calls
- Well known from browsers for javascripts, java VMs and the like

Methods of JITing considered

- Emit C code, invoke compiler, generate shared object, dlopen()
  - requires a lot of forking
  - requires C compiler
  - doesn’t easily allow inlining
- Directly emit machine language, remap memory executable
  - fastest to emit
  - no optimization (including inlining)
  - lots of per-architecture work
  - very few people want / able to maintain
  - fun
- Use compiler / optimizer framework with JIT support
  - issues around licensing, portability, maturity
  - JIT often not most common user
- => LLVM

EDB POSTGRES™

LLVM

- Compiler Framework
- Intermediate Representation
  - can be generated for C code using clang!
- Optimizations
- JIT Support
- https://llvm.org/
- Used among my others by
  - clang C, C++ compiler
  - swift
  - rust
  - other database like products
  - ...

Postgres LLVM usage

- C vs. C++
- LLVM usage in shared library
  - can be installed separate from main postgres package
  - C++ usage encapsulated
- Error handling
- Inlining Support

v10+ Expression Evaluation Engine

- WHERE a.col < 10 AND a.another = 3
  - EEOP_SCAN_FETCHSOME (deform necessary cols)
  - EEOP_SCAN_VAR (a.col)
  - EEOP_CONST (10)
  - EEOP_FUNCEXPR_STRICT (int4lt)
  - EEOP_BOOL_AND_STEP_FIRST
  - EEOP_SCAN_VAR (a.another)
  - EEOP_CONST (3)
  - EEOP_FUNCEXPR_STRICT (int4eq)
  - EEOP_BOOL_AND_STEP_LAST (AND)
- direct threaded
- lots of indirect jumps

```
EEO_CASE(EEOP_FUNCEXPR_STRICT)
{
    FunctionCallInfo fcinfo = op->d.func.fcinfo_data;
    bool    *argnull = fcinfo->argnull;
    int             argno;
    Datum           d;

    /* strict function, so check for NULL args */
    for (argno = 0; argno < op->d.func.nargs; argno++) // unnecessary
    {
        if (argnull[argno])
        {
            *op->resnull = true;
            goto strictfail;
        }
    }
    fcinfo->isnull = false; // optimized away
    d = op→d.func.fn_addr(fcinfo); // indirect
    *op->resvalue = d;  // moved to register
    *op->resnull = fcinfo->isnull;

strictfail:
    EEO_NEXT();  // indirect, optimized away
}
```

JITed expressions

- directly emit LLVM IR for common opcodes
- emit calls to functions implementing less common opcodes
  - can be inlined
- indirect opcode → opcode jumps become direct
- indirect funcexpr calls become direct
  - can be inlined
- TPCH Q01 non-jitted vs jitted:
  - 28759 ms vs 22309 ms
  - branch misses: 0.38% vs 0.07%
  - iTLB load misses: 58,903,279 vs 48,986 (yes, really)

```
block.op.2.start:                              ; preds = %block.op.1.start
  %v_argnullp = getelementptr inbounds %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %v_fcinfo, i32 0, i32 7
  store i8 1, i8* %resnullp
  br label %check-null-arg

check-null-arg:                                ; preds = %block.op.2.start
  %25 = getelementptr inbounds [100 x i8], [100 x i8]* %v_argnullp, i32 0, i32 0
  %26 = load i8, i8* %25
  %27 = icmp eq i8 %26, 1
  br i1 %27, label %block.op.3.start, label %check-null-arg1

check-null-arg1:                               ; preds = %check-null-arg
  %28 = getelementptr inbounds [100 x i8], [100 x i8]* %v_argnullp, i32 0, i32 1
  %29 = load i8, i8* %28
  %30 = icmp eq i8 %29, 1
  br i1 %30, label %block.op.3.start, label %no-null-args

no-null-args:                                  ; preds = %check-null-arg1
  %v_fcinfo_isnull = getelementptr inbounds %struct.FunctionCallInfoData,
%struct.FunctionCallInfoData* %v_fcinfo, i32 0, i32 4
  store i8 0, i8* %v_fcinfo_isnull
  %funccall = call i64 @date_le_timestamp(%struct.FunctionCallInfoData* %v_fcinfo) #13
  %31 = load i8, i8* %v_fcinfo_isnull
  store i64 %funccall, i64* %resvaluep
  store i8 %31, i8* %resnullp
  br label %block.op.3.start
```

EDB POSTGRES

```
Samples: 87K of event 'cycles:ppp', cnt (approx.): 71706618234
  Overhead  Command   Shared Object     Symbol
-   35.96%  postgres  postgres          [.] ExecInterpExpr
      + 72.86% ExecAgg
      - 18.33% tuplehash_insert
           LookupTupleHashEntry
           ExecAgg
           ExecSort
      + 8.81% ExecScan
-   10.79%  postgres  postgres          [.] slot_deform_tuple
        slot_getsomeattrs
      - ExecInterpExpr
         + 77.31% ExecScan
         + 22.69% tuplehash_insert
+    4.96%  postgres  postgres          [.] tuplehash_insert
+    4.53%  postgres  postgres          [.] float8_accum
+    3.21%  postgres  postgres          [.] float8pl
+    2.61%  postgres  postgres          [.] bpchareq
+    2.40%  postgres  postgres          [.] hashbpchar
```

Tuple Deforming

- deforming := turn on-disk tuple into in-memory representation
- Often most significant bottleneck
- TupleDesc (“tuple format”) can be made known at JIT time in many cases
- Optimizable:
  - Number of columns to deform - constant
  - Number of columns in tuple – if to-deform below last NOT NULL
  - column type - constant
  - column width – known for fixed width types
  - Variable alignment requirements – known for fixed width (depending on NULLness)
  - NULL bitmap – no need to check if NOT NULL
- Resulting code often very pipelineable, previously lots of stalls
- Access to tuple’s `t_hoff` / `HeapTupleHeaderGetNatts()` still major source of stalls
- TPC-H Q01: unjitted deform vs jitted
  - time: 22277 ms vs 19580 ms
  - branches: 1396.318 M/sec vs 1161.628M/sec (despite higher throughput)

Inlining

```
CREATE OPERATOR pg_catalog.= (
    PROCEDURE = int8eq,
    LEFTARG = bigint,
    RIGHTARG = bigint,
...
);
CREATE OR REPLACE FUNCTION pg_catalog.int8eq(bigint, bigint)
    RETURNS boolean
    LANGUAGE internal
    IMMUTABLE PARALLEL SAFE STRICT LEAKPROOF
AS $function$int8eq$function$
```

```
Samples: 87K of event 'cycles:ppp', cnt (approx.): 71706618234
  Overhead  Command   Shared Object      Symbol
-   35.96%  postgres  postgres           [.] ExecInterpExpr
      + 72.86% ExecAgg
      - 18.33% tuplehash_insert
           LookupTupleHashEntry
           ExecAgg
           ExecSort
      + 8.81% ExecScan
-   10.79%  postgres  postgres           [.] slot_deform_tuple
        slot_getsomeattrs
      - ExecInterpExpr
         + 77.31% ExecScan
         + 22.69% tuplehash_insert
+    4.96%  postgres  postgres           [.] tuplehash_insert
+    4.53%  postgres  postgres           [.] float8_accum
+    3.21%  postgres  postgres           [.] float8pl
+    2.61%  postgres  postgres           [.] bpchareq
+    2.40%  postgres  postgres           [.] hashbpchar
```

Inlining

- All operators in postgres are functions! Lots of external function calls
- Postgres function calls are expensive, lots of memory indirection
- Convert sourcecode to bitcode at buildtime, install into

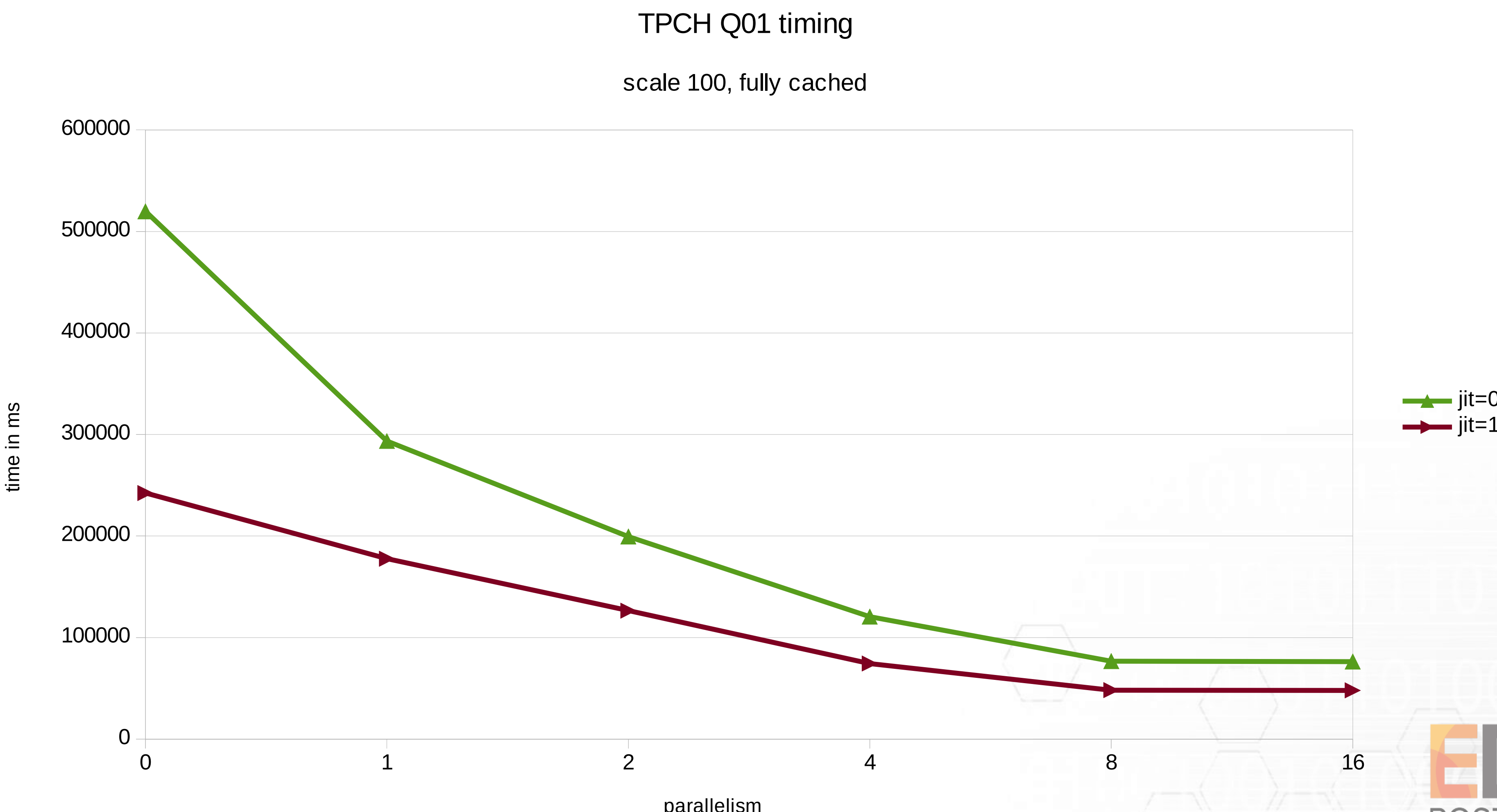
  `$pkglibdir/bitcode/<module>.index.bc`

  `$pkglibdir/bitcode/<module>/path/to/file.bc`
- LLVM’s cross-module inlining not suitable
  - requires exporting of symbols at compile time, unknown which needed
- Postgres specific inlining logic:
  - lookup symbol in summary corresponding to function
  - inlining safety check (no mutable static variables referenced)
  - cost analysis
  - inline function, referenced static functions, referenced constant static variables (mainly strings)
  - use `llvm::IRMover` to move relevant globals
  - can’t cache modules in memory, cloning expensive and incomplete
- Avoids need to implement direct JIT emission for lots of semi critical code
- Function call interface significantly limits benefits

Planning JIT

- Naive!
- Perform JIT if `query_cost > jit_above_cost`
- Optimize if `query_cost > jit_optimize_above_cost`
- Inline if `query_cost > jit_inline_above_cost`
- Whole query decision
- *NOT* a tracing JIT:
  - costing makes tracing somewhat superflous
  - tracing decreases overall gains

Faster Execution:
JIT Compilation



JIT Issues – Code Generation

- Expressions refer to per-query allocated memory
  - generated code references memory locations
  - optimizer can't optimize away memory lots of memory references
  - FIX: separate permanent and per eval memory
- Function Call Interface requires persistence
  - **lots** of superflous memory reads/writes for arguments, optimizer can't eliminate in most cases
    - massively reduces benefits of inlining
  - FIX: pass FunctionCallInfoData and FmgrInfo separately to functions
    - remove FunctionCallInfoData->flinfo
    - move context, resultinfo, fncollation to FmgrInfo
    - move isnull field to separate argument? Return struct?
- Expression step results refer to persistent memory
  - move to temporary memory

EDB POSTGRES

JIT Issues - Caching

- Optimizer overhead significant
  - TPCH Q01: unopt, noinline: time to optimize: 0.002s, emit: 0.036s
  - TPCH Q01: time to inline: 0.080s, optimize: 0.163s, emit 0.082s
- references to memory locations prevent caching
- Introduce per-backend LRU cache of functions keyed by hash of emitted LRU (plus comparator)
  - relatively easy task
- Allow expressions to be generated at plan time, and tied to a prepared statement
  - medium – hard

EDB POSTGRES

JIT Issues – Planning

- Whole Query decision too coarse
  - use estimates about total number of each function evaluation?
- Some expressions guaranteed to only be evaluated once
  - VALUES()
  - SQL functions

Future things to JIT

- COPY input / output
  - easy – medium
- Aggregate & Hashjoin hash computation
  - easy
- in-memory tuplesort
  - easy
- Whole of Executor
  - wheeee

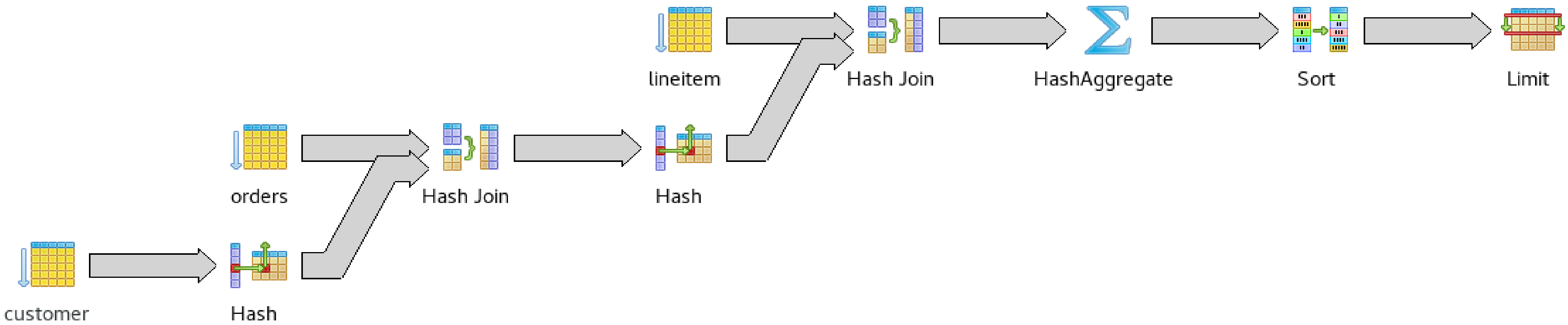
Readonly OLTP: per query overhead

- ```
  SELECT abalance
  FROM pgbench_accounts
  WHERE aid = $1;
  ```

```
-   73.05%     2.16%  postgres                       [.] PostgresMain
       - 29.71% exec_bind_message (inlined)
          - 51.63% PortalStart
             + 90.95% standard_ExecutorStart
          + 10.77% GetCachedPlan
...
       + 29.66% exec_execute_message (inlined)
       + 13.80% finish_xact_command
```

- => Move work from executor => planner
- => Reduce overhead by using smarter datastructures
```

Analytics: Batched query execution



Overhead:

- repeated Buffer Lookup / Locking / Pinning overhead
- Poor data cache locality
- inefficient use of CPU pipeline

Improving Postgres' Efficiency using JIT and other techniques

Andres Freund

PostgreSQL Developer & Committer

Email: andres@anarazel.de

Email: andres.freund@enterprisedb.com

Twitter: @AndresFreundTec

anarazel.de/talks/scale16x-2018-03-09/jit-and-other-efficiency.pdf